Heterogeneous **Client** Programming

Graphics & OpenCL

*(with the luxury of hindsight)*

*David Blythe*

*Intel*
Overview

• Blythe Intro
• Definitions
• Short history of graphics parallel programming
• Critique of GPGPU
• Future
I like to think about “global architecture”

Microsoft Windows Graphics Stack circa 2007  (source: Blythe)
Think about: Top Down vs. Bottom Up Design

• Top down -> portable, usable abstractions with implementation latitude
  – E.g., in-order pipeline with out-of-order implementation

• Bottom Up -> abstractions that reflect hw implementation/architecture choices
  – NUMA, non-coherent caches
  – VGA register interface is not a good abstraction
    • VGA compatibility still haunts the PC graphics industry
Market Segmentation

• Lots of ways to slice the pie
  – Data Center
    • HPC, Cloud, ....
  – Workstation
  – Client
    • Desktop
    • Laptop/Notebook
    • Netbook/Slate
    • Ultramobile (handheld)
  – Embedded
    • Automotive, Point of Sale, ...
Client characteristics

• Single socket (or socket + I/O devices)
  – Long term everything is SoC
  – Don’t sweat over multi-socket, clusters, ...

• Large volume of non-specialist programmers
  – einstein, elvis, mort persona space
  – cf. HPC addresses complexity problems by including application engineer with the machine

• End customer cares about the final experience
  – 100’s of millions of end customers
  – most of the interesting experiences involve pixels
  => experiences need to be easy-ish to produce
GPU Success Story

• Visual experiences are compelling
  – High demand for more sophisticated experiences => aggressive evolution

• Successful API abstractions
  – “3D pipeline”
  – Portable, stable, easy-ish to use, ...
  – “linear” evolution of APIs (evolution vs. revolution)

• Happy coincidence
  – Most successful parallel programming model (to date)
    • Metrics: # programmers, # devices, ....
  – Not something that was carefully planned out 😊
## Parallel Programming & Graphics

<table>
<thead>
<tr>
<th>Multi-processor mainframes/mincomputers/workstations – e.g., Raytracing</th>
<th>1980s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom VLSI (Silicon Graphics, ...) Pipelined parallelism, e.g., geometry engine Parallel pipelines (cf. sort first, middle, last)</td>
<td>1980s</td>
</tr>
<tr>
<td>SIMD/tiny-vector processing for vertices/pixels Commodity parts &amp; ASICs</td>
<td>1990s</td>
</tr>
<tr>
<td>Developer-exposed programmability (shaders)</td>
<td>2000-2006</td>
</tr>
<tr>
<td>GPGPU Generalization of shader capabilities (flow control, integers, ld/st, ...)</td>
<td>2007+</td>
</tr>
<tr>
<td>Heterogeneous programming</td>
<td>2011+</td>
</tr>
</tbody>
</table>
Select GPU Evolution details

- Why is/was the pipeline the way it was
- What changed

+ free editorial commentary
Mid-1990s Pipeline Characteristics

- Async pipeline, no/minimal read back
  - Fire & forget (result goes to display)
  - Allows deep pipeline, buffering, overlapped CPU execution, “add-in” card model
- Non-CPU accessible framebuffer, textures
  - Allows replication, data layout transforms, …
- In-order pipeline
  - Implementations can go temporarily out-of-order
- Immutable, non-CPU accessible display lists
  - Allowed hw-specific implementations, minimal book keeping
- CPU mutable geometry data (vertices, vertex attributes)
  - Caused implementations to do nasty hacks to allow caching
- Abstraction decoupled from implementation:
  - SW API/driver model that supports multiple implementations (simultaneously) => Rich ecosystem
  - Mixture of fixed-function and (unexposed) programmable elements
Changes to 1990s Pipeline

• Async pipeline, no/minimal read back
  – Fire & forget (result goes to display)
  – Allows deep pipeline, buffering, overlapped CPU execution, “add-in” card model
  – Major tension point (on-die integration?)

• Non-CPU accessible framebuffer, textures
  – Allows replication, data layout transforms, ...
  – Restriction replaced with mine-yours access (release consistency)

• In-order pipeline
  – Implementations can go temporarily out-of-order

• Immutable, non-CPU accessible display lists
  – Allowed hw-specific implementations, minimal book keeping
  – Concept added to DX11, removed from OpenGL ES

• CPU mutable geometry data (vertices, vertex attributes)
  – Caused implementations to do nasty hacks to allow caching
  – Replaced with mine-yours access (and the ever-popular NO_OVERWRITE)

• Abstraction decoupled from implementation:
  – SW API/driver model that supports multiple implementations (simultaneously) => Rich ecosystem
  – Mixture of fixed-function and (unexposed) programmable elements
Other 1990s pipeline badness

• Slow pipeline state changes (texture change)
  => batching added to the vernacular
  – Batching affects application structure adversely
  – Add “instancing” to turn state change into an “indexing problem”
    • Every problem can be solved by adding a level of indirection

• State machine model too unwieldy (increased flexibility)
  – For programmer:
    • “Register combiners” for multi-texture composition
    • Straw that broke the camels back?
      => shaders
  – For pipeline implementer:
    • State updates too fine-grain
      => refactor state into “state objects”

• Too many optional features (hard to write portable programs)
  => remove optional features
Mid-2000s (shader) characteristics

- Separate specialized/typed memories
  - Constant buffers, scratch, buffers, textures, render targets
  - Optimization of cache structures (read only, uniform access, ...)
- Controlled “side effects” to memory (mid-pipe stream out, pixel writes)
  - Allow replay-based context switching, vertex shader caching, ...
- No simultaneous read/write access to a resource
  - E.g., read texture & write to it as render target
  - Determinism, implementation optimizations
- No scatter (writes go to pixel location determined by rasterization, or stream out)
  - Implementation optimizations, performance
- No cross-item (vertex, pixel) communication
  - Scheduling optimizations, simplicity
- No atomic ops, sync ops
  - Performance 😊
Changes to Mid-2000s pipeline)

• Separate specialized/typed memories
  – Constant buffers, scratch, buffers, textures, render targets
  – Optimization of cache structures (read only, uniform access, ...)
  – Remove: ????

• Controlled “side effects” to memory (mid-pipe stream out, pixel writes)
  – Allow replay-based context switching, vertex shader caching, ...
  – R/M/W everywhere ????

• No simultaneous read/write access to a resource
  – E.g., read texture & write to it as render target
  – Determinism, implementation optimizations
  – Remove???? - determinism is so 2000s

• No scatter (writes go to pixel location determined by rasterization, or stream out)
  – Implementation optimizations, performance
  – Add scatter (add load/store)

• No cross-item (vertex, pixel) communication
  – Scheduling optimizations, simplicity
  – R/M/W to “shared local memory” and “global memory”

• No atomic ops, sync ops
  – Performance 😊
  – Add atomics, barriers (fences too)
More shader-pipeline badness

- JIT compilation model
  - Computationally expensive to compile from source
  - Difficult to build a robust caching system
    - cf. .net “gac”
    - Increase complexity with upgradeable/removable GPUs

- Lack of standardized compiler (front ends)
  - Compilers are hard (front ends too)
    - Not really that many C/C++ compilers, fewer front ends
  - Poor developer experience with inconsistent implementations

=> Microsoft HLSL produces “standard” intermediate representation
Idea

• Hmm, maybe we could apply this graphics programming model to other things

or

• When all you have is a GPU, everything looks like a pixel

=> enter OpenCL (and DirectCompute) as “Compute” APIs
“Compute” API Design

What parts to keep, what parts to “improve”?

• Remove graphics concepts
  – Rasterizing a primitive to launch work
  – Vertices, primitives, pixels, ...
  – Complex 3D pipeline
  – Graphics API interop?

• Keep
  – Shader/kernel concept
  – 1D,2D,3D inputs, outputs to memory

• Rename
  – Draw*-> NDRange
  – Pixel -> Work Item
  – Texture -> Image

• Add
  – Shared local memory
  – Atomic operations, barriers
  – Events
  – Multi-device contexts (e.g., CPU+GPU devices)
  – Lots of implementation characteristics to query
  – Work item, work group IDs
OpenCL Execution Model

(source: OpenCL 1.2 spec)
OpenCL Conceptual Device Architecture

(source: OpenCL 1.2 spec)
Local Memory (LM)

• Allow multiple SIMD elements (work items) to cooperate on a data structure
  – Efficient gather & scatter to scratchpad
• Efficiency requires compromise
  – Not all cores can share a single LM (scalability)
• Why?
  – How many disjoint memory accesses/clock?
  – Most memory systems are cache-line oriented
  – How many distinct cache lines reads/clock?
Ripple Effects of LM

- Exposed to application programmer:
  - compute unit, work group, work group size
  - queries added to API
    - Need to inspect kernel to determine WG size
- Implementation constraint
  - WG and LM are scheduled together
  - Gang scheduling

- Q: when should a programmer use LM versus global memory?
  - What if there is a local cache?
  - What about structure-of-array vs. array-of-structure data layout?
Adding a local cache

Compute Device

Compute unit 1

Private memory 1

PE 1

Local memory 1

... ...

Private memory M

PE M

Local R/W Cache

Compute unit N

Private memory 1

PE 1

Local memory N

Local R/W Cache

Global/Constant Memory Data Cache

Global Memory

Constant Memory

Compute Device Memory
Atomics & Barriers

• What could go wrong?
  – How are atomics implemented?
    • Local vs. global memory
    • Implemented in core or as remote (memory-side ops)
  – If a programmer cares about performance:
    • They will match algorithm to the implementation
      – Not unique to compute, happens for graphics too
  – How much parallelism is really achieved?
The “CPU” Device

• Lets allow OpenCL kernels to run on CPUs
  – Seems like a good idea, esp. if no GPU present

• But, ...
  – CPU has different characteristics

• Temptation to convert async model to sync model
  – E.g., sync kernel execution, sync callbacks, ...

• Scatter/gather support & local memory?
  – Write algorithms to match implementation
Multi-Device Context

• E.g., allow CPU and GPU device to share buffers, images, programs,

• Does it really solve a problem?
  – What if images have different tiling transforms
    • Need to convert back/forth for CPU/GPU access

• Should a multi-device context with 2 different GPU manufacturers work?
Does OpenCL Provide Enough Abstraction?

Intel, nVidia, AMD SIMD execution models  (source: realworldtech.com)
Future Enhancements (OpenCL 2.0)

- Tasking
- Fixed function integration/exposure
- Load sharing
- Exposed intermediate language
Tasking

• Multicore CPU programmers adopting tasking systems (task-oriented parallelism)
  – Boost, TBB, ConcRT, ....
  – Break work into small tasks and let task system schedule/load balance

• Put it on the GPU too?
  – OpenCL 1.0 has degenerate “task”
  – 1 work item NDRRange

• Enhance this with better syntax
  AND

• Allow a kernel to submit new work
Tasking Complexities

• What is granularity of task?
• How does a task map to a hw-thread & core
  – E.g., task runs at granularity of 1 hw-thread
• How does task scheduling interact with “gang scheduling” threads in a work group?
  – Do tasks interfere with work groups
• How should task spawning work?
  – Spawn general NDRanges, tasks?
  – Does it need hw scheduling/spawning support
    • Using a GPU core to execute scheduling code seems inefficient
    • Round trip through CPU/driver?
  – Need to analyze real workloads to answer these questions
Fixed-Function Integration/Exposure

• Fixed-function for power efficiency
  – Use for “well understood” primitives
  – Already include in kernel language as intrinsic function
    • E.g., texture sampler
• What if “work item granularity” isn’t right?
  – E.g., operate on a block of pixels (input n x m, output n x m)
  – Change effective work group size with conditionals on work item IDs
  – OR integrate into task framework
• What about operations that aren’t suitable for invoking from a kernel?
  – Very coarse grain, don’t fit into kernel abstraction ....
  – Could put in a different API and do API interop
  – OR expose as “predefined” kernels
Exposed Intermediate Language (IL)

• Separate the front end compiler/language evolution from the execution engine
  – E.g., support C, C++, Haskell, ...
  – cf. ptex, FSAIL, ...

• How low to go?
  – What problem is being solved (requirements)?
    • A way to avoid shipping source code with app?
    • A way to avoid expensive JIT?
    • A more stable/general code generation target?
    • A **portable** version of one or more of the above?
Load Sharing

• Q: Is there an opportunity to use both CPU and GPU devices simultaneously?
• A: It depends
  – Is there sufficient power/thermal headroom for both?
  – Trend for low power devices, doesn’t look promising

• Need to distinguish homogeneous and heterogeneous load sharing
  – E.g., data parallel vs. single thread/latency sensitive code
  – Heterogeneous load sharing seems pretty interesting
  – Don’t necessary run in parallel – use best processor for “task”
  – Do OpenCL abstractions help for the whole hetero workload?
Summary

• GPUs have come a long way
• Compute hasn’t really proven itself
  – At least not on client
• Real challenges around portable abstractions
• Programming model is awkward
  – Abstractions, lack of language unification
• Lots of additions being proposed
  – Not clear we are building on the right foundation
=> Lots of exploration left to do